PHASE 2 DESIGN DOCUMENT

By: Shaul, Andy, Rachel, Yash, Elliot, Shridhar

Updated Specification: Our group is creating a Scrabble game that allows two or more human players, or AI players, to play against each other. Broadly, the Scrabble board is a 15 x 15 board of squares; some of the squares are equipped with certain special bonuses that will alter how a word that a player plays is calculated. When it is a player's turn, they can choose from one of three options: playing a word on the board, reshuffling their rack by putting some tiles back in the bag and redrawing, or passing their turn. If a word is placed on the board by a player, another player can choose to challenge the word if they believe the word is invalid. If there is a challenge, we check if it is a valid play by searching for the word in the Scrabble dictionary; if it is a valid play, the player who challenged the word loses their turn; if not, the word is removed from the board and the player who originally played the word does not receive the points for the word. A game of Scrabble ends when there are no letter tiles remaining in the bag, or the players have run out of tiles and valid moves to make. Then, the player with the highest total number of points after subtracting the points of the tiles remaining on their rack is the winner.

Major Design Decisions: Originally for rendering the different components (board, tiles, squares, etc), we had the code structured so that each of these classes had its own render method. Upon further thought, we realized that structuring the code this way seemed to violate the Single Responsibility principle, since these classes were now additionally responsible for their own rendering. We also realized that this design would violate clean architecture as the rendering methods are part of the Interface Adapters layer, whereas the classes they were being implemented in were otherwise Entities. We decided that it would be better to create a Renderer class, responsible for all the rendering, and the code was refactored so that these rendering methods became part of the new Renderer class.

Clean Architecture: We have designed our classes, so that each falls under exactly one of the clean architecture layers. As explained above, this was still an issue until rendering was refactored. Our refactoring in general helped adhere more strongly to the clean architecture. By splitting some classes from phase 0 into several smaller classes we were able to more strongly demonstrate the division between the different layers. We also made sure to adhere to the clean architecture within our testing, by testing the business rules without relying on any external elements. We also maintained UI independence with the introduction of GUI.

While we did take careful steps to ensure clean architecture in the inner layers, there are some violations of clean architecture in the outer layers, such as dependencies going through more than one layer, and one location where the dependency needs to be inverted. This was mostly because these classes were added last minute, and we insufficient refactored these classes.

SOLID Design Principles: We have tried to adhere to the SOLID principles as much as possible. For example, we have different classes for each different part of the game (such as Bag, Board, PlayerManager) which adheres to the Single Responsibility Principle: each of these classes has

only one reason to change. We also adhere to the Open-Closed Principle; having a Move interface allows for an easy extension of adding different types of moves, which would be useful if we wanted to create an "enhanced" version of Scrabble with some other types of moves.

Packaging Strategies: As suggested in the lecture on packages, we mainly considered two packaging strategies: packaging by layer and packaging by component. Ultimately, we decided to use packaging by component, and decided to create one outer package encompassing everything, including the class containing our main method, and two additional packages within this package: one for the internal game data (ex. Tiles) and one for the GUI related functions. We used this packaging strategy since we thought it organized our classes in the cleanest manner and reflected encapsulation well. Furthermore, packaging by component allows classes that are coupled together to be in the same package, reducing the need for public methods and allowing us to make better use of package-private methods.

Design Patterns: *Command Design Pattern:* We used the Command design pattern to represent the different types of moves. The Command design pattern consists of an Order interface, which acts as a command, and concrete Command classes that implement the Order interface. It also consists of a Receiver, which places orders. In this case, the Move interface is the Order interface, and the PassMove, PlaceMove, and SwapMove classes correspond to the concrete command classes that inherit from the interface, and the main game loop places the orders for the moves. Our use of this design pattern is further highlighted by the use of the execute method signature in the Move interface.

Memento Design Pattern: We used the Memento design pattern for the GameState. The Memento design pattern consists of a Memento class, which contains the state of the object to be restored, the Originator class, which creates and stores Memento objects, and the CareTaker class, which restores object states from Memento. In this case, the GameState class is the Memento, the Game class is the Originator, and the ScrabbleGame class is the CareTaker. Indeed, we see that the methods in these classes are consistent with the design pattern; for example, the loadGameState and getGameState methods in Game correspond to the Originator loading and saving the Memento.

(Note: the following links were used to describe the design patterns: https://sourcemaking.com/design_patterns/command, https://www.tutorialspoint.com/design_pattern/memento_pattern.htm).

Progress Report:

Name	Contribution Since Phase 1	Link to Significant Pull Request
Shaul	Creating and implementing AI players of various difficulties.	https://github.com/CSC207-UofT/course-project-windrunners/pull/54

		The commits in this show the creation of AI players and and how it was integrated into the game. An AI player is used to allow for single player gameplay.
Andy	Implementing challenges to CLI (where another player contests whether a word placed on the board is valid) Refactored to more specific packages Fixed bugs in AI merge with main game loop	https://github.com/CSC207-UofT/course-project-windrunners/pull/59 This commit shows when we merged the challenges into the cli, as well as the package and ai changes.
Rachel	Fixing GameState bugs, writing GameState class tests, significant portions of presentation and written documents	https://github.com/CSC207-UofT/course-project-windrunners/pull/34 The commits in this PR demonstrate show the creation of the GameState class and its development into a fully functional class.
Yash	Integrated Place Move, Swap move, and Pass move with the GUI; updated class Board to find the word formed by tiles placed as part of a move; cancelling a swap move midway; remove tiles from board that were inserted as part of a Place move; wrote tests.	Add GUI Input Handling, including the different types of moves by elliotschrider · Pull Request #52 · CSC207-UofT/course-project-windrunners (github.com) The commits in this PR show how the GUI was integrated into the existing code. Additional methods were added to deal with placements of words/swapping tiles/passing move directly from the GUI. GameState was used to restore the GUI in case of an invalid move.
Elliot	Added GUI window and method for rendering various components. Added ability to click and place Tiles on the board. Refactored GUI code.	https://github.com/CSC207-UofT/course-project-windrunners/pull/21 Add GUI Input Handling, including the different types of moves by elliotschrider · Pull Request #52 · CSC207-UofT/course-project-windrunners (github.com)

Shridhar	Adding wildcards	https://github.com/CSC207-UofT/course-project-windrunners/pull/51

Note on Testing

- As we mentioned in our previous discussions with you, we asked on Piazza whether we had to test the GUI and we heard that it was optional. So, our code does not include a test suite for the GUI.
- We had a discussion about testing the AI player with you and concluded that not much could be done other than testing that the AI makes a valid move

ACCESSIBILITY REPORT

1. **Equitable Use**
   - We provide the same means of use for all users. When it is their turn, each user sees their rack, the tiles on the board, and the number of points that each player has.
   - Our GUI looks very similar to the real Scrabble board, and the user of colours to distinguish between special squares and normal squares on the board, for example, makes our design visually appealing for all users.
   - Despite this, our design may segregate those who do not wish or are not able to perceive information visually; currently, to know what tiles are on the board, there is no way to do so other than looking at our CLI or GUI.

2. **Flexibility in Use**
   - Our design provides choice in methods of use; for instance, users have the choice between playing using the CLI (although we eventually decided to limit this to focus on the GUI) and the GUI.
   - The tiles on a player's rack and the squares on the screen are a reasonable size – most users should be able to click on the tile they want to place and the square they want to place it on without much difficulty. Furthermore, our GUI is much more intuitive and allows users to interact with the board directly (rather than through battleship coordinates, for instance), which makes the user much less prone to error.
   - There is no time limit on a user's turn – the user declares when they are finished with their move – which provides adaptability to the user's pace.

3. **Simple and Intuitive Use**
   - The GUI has a very intuitive design in that it looks very similar to the real Scrabble board. Placing tiles on the board (by clicking on the tile and the square to place the tile on) is very intuitive and similar to actual Scrabble.
   - Our program only allows English Scrabble to be played. A next step would be to accommodate other languages by adding Scrabble dictionaries for other languages and additional tiles, if needed (such as accented characters).
   - The GUI indicates which user's turn it is – if a user has completed their turn successfully, they will see another user's name pop up. However, this design could be made more intuitive – for example, by adding text that says "Success!" after a user has successfully completed their turn.

4. **Perceptible Information**

- Currently, our program only presents information in a visual manner. Unfortunately, this poses barriers for people who are visually impaired. One step we could take is to allow a text-to-speech option to communicate the information on the Scrabble board and rack. This would increase the "legibility" of essential information.
- Our design also poses another barrier – there are no ways of placing tiles on the board except through using your hands and a mouse or keyboard. Another step we could take to remove this barrier is to allow users to give verbal commands to place tiles.

5. **Tolerance for Error**
   - Our design allows users to go back and place the tile back on their rack if they make a mistake while placing a word on the board, tolerating for user error
   - There are no physical hazards from our program. We also did not include any loud, sudden noises or flashes, which may be hazardous to people with certain medical conditions.

6. **Low Physical Effort**
   - Our game can be played while sitting down, using a mouse and keyboard, so it is not particularly strenuous
   - There is also no time limit for any game of Scrabble, allowing people to take a break if they are tired. This minimizes sustained physical effort.

7. **Size and Space for Approach or Use**
   - The important elements (the rack, board, player names, and scores) fit on the screen
   - However, our program may not be compatible with personal assistance devices that are installed on computers. A next step would be to explore this integration.

We would market our program towards people who enjoy Scrabble and want to enjoy it in a wider variety of settings. The traditional version of Scrabble is quite clunky – requiring space for a board and tiles. This makes playing Scrabble in confined settings, such as on public transportation, more difficult. It also requires that a player remember to bring all the equipment required for the game. Our version of Scrabble is much more accessible and easier to carry around, and thus much more appealing.

Our program relies highly on vision – the essential information about a Scrabble game is conveyed to the user in a visual manner, whether this be through the CLI or the GUI. Thus, visually impaired persons are much less likely to use our program. Additionally, our Scrabble

game can only be played in English, and the prompts to the users (such as asking for the number of players) are given in English. Thus, non-English speaking people are less likely to use our program.